Parallel Programming with Declarative Ada

John Thornley
Computer Science Department
California Institute of Technology
Pasadena, California 91125, USA
john-t@cs.caltech.edu

April 24, 1993

Abstract

Declarative programming languages (e.g., functional and logic programming languages) are semantically elegant and implicitly express parallelism at a high level. We show how a parallel declarative language can be based on a modern structured imperative language with single-assignment variables. Such a language combines the advantages of parallel declarative programming with the strengths and familiarity of the underlying imperative language. We introduce Declarative Ada, a parallel declarative language based on a subset of Ada. Declarative Ada integrates parallel and sequential composition, allowing sequential input and output from within parallel declarative programs.

1 Introduction

Goal

Our goal is to investigate the feasibility of basing a parallel declarative programming language on a conventional imperative language. The requirement that is fundamental to declarative programming is the single-assignment restriction, prohibiting multiple assignments to the same variable. Programs with this restriction implicitly express parallelism at all

levels of granularity. In most other respects, parallel declarative programs can be identical to structured imperative programs. As the basis of our investigation, we have designed and implemented Declarative Ada, a parallel declarative language based on a subset of Ada [1].

Ada

Ada is an imperative language that supports modern software engineering principles. Program reliability and maintenance, and programming as a human activity were among the overriding concerns in the design of Ada [1, Section 1.3]. Features that influenced the choice of Ada as the basis of our investigation include:

- Syntax that avoids error-prone notations.
- Explicit declaration and type specification of all variables.
- Strong typing and no implicit type conversions.
- Structured language constructs.
- Small number of underlying concepts integrated in a consistent way.
- Compile-time error checking.
- Compile-time enforcement of intended programming practice.
- Stable and well-described language definition.
- Readable to any person familiar with C, Fortran-77, or Pascal.

An excellent introduction to software engineering principles, the goals and development of Ada, the Ada language, and Ada programming practice is given by Booch [2].

Explicit Parallelism in Ada

Standard Ada is already a parallel programming language. The Ada *tasking model* is based on the concept of communicating sequential processes. A

task is a program unit that is explicitly executed in parallel with other program units. Communication and synchronization is achieved by *rendezvous* between a task issuing an *entry call* and a task *accepting* the call. Tasking is well suited to expressing parallelism in problems with specifications that explicitly involve concurrency, e.g., controlling or simulating physically distributed systems. However, tasking is less suited to expressing parallelism in problems with specifications that do not explicitly involve concurrency, e.g., scientific supercomputing. For these problems, explicit specification of tasking is an extra layer of detail that makes parallel programs usually more complicated and less portable than sequential programs.

Declarative Ada

The syntax of Declarative Ada is a subset of the syntax of Ada. The language includes: procedures and functions; assignment, if-then-else, for-loop, procedure-call, and function-return statements; integer, floating-point, Boolean, array, record, and access (pointer) types and operators. The significant difference between Declarative Ada and standard Ada is that all Declarative Ada variables are single-assignment variables. Most other restrictions are because Declarative Ada is intended as a small demonstration language. An extended version of Declarative Ada could be based on the full Ada syntax.

Implicit Parallelism in Declarative Ada

Declarative Ada does not include tasking. Parallelism is expressed implicitly, as a consequence of the single-assignment restriction. Executions of statements, and evaluations of variables and expressions are implicit processes. Processes can be executed in any order in which variables are assigned values before they are used in expressions. Data flow, rather than textual sequence, guides the order of execution, and independent processes can be executed concurrently. Program results are deterministic and repeatable, regardless of the number of processors and the scheduling of processes. Since parallelism is implicit, parallel Declarative Ada programs are no more complicated and no less portable than sequential Ada programs. Many parallel Declarative Ada programs can be correctly executed as sequential Ada programs without modification.

Integrating Parallel and Sequential Composition

An extended version of Declarative Ada could integrate single-assignment and multiple-assignment variables, and parallel and sequential composition. Multiple-assignment variables could be used in circumstances where they allow for more simple or more efficient programs, and explicit sequential composition could be used for ordering the execution of operations on multiple-assignment variables. Declarative Ada already includes sequential composition, for ordering the execution of input and output statements. Sequential input and output files are a specific instance of multiple-assignment variables.

Integrating Explicit and Implicit Parallelism

The Declarative Ada model of parallelism is compatible with the Ada tasking model. An extended version of Declarative Ada based on the full Ada syntax could integrate explicit parallelism from tasking and implicit parallelism from single-assignment variables. Tasking could be used for the parts of a problem that explicitly involve concurrency, and implicit parallelism from single-assignment variables could be used for the parts of a problem that do not explicitly involve concurrency.

An Implementation of Declarative Ada

A prototype implementation of Declarative Ada [11] is available from the author. The implementation—written entirely in ANSI C—compiles Declarative Ada into PCN [6], a parallel programming language that is freely available for many computer systems, including most Unix-based workstations. The implementation has been used by students at Caltech and University of Auckland.

2 Declarative Ada

In this section we summarize Declarative Ada and describe its support for parallel programming. The complete language definition is given in [12].

2.1 Language Summary

Program Structure and Declarations: A program consists of a sequence of constant, type, and subprogram (procedure and function) declarations. One of the subprograms is executed as the *main program*. Parameters of procedures can be of **in**, **out**, and **in out** mode. Parameters of functions must be of **in** mode. Subprograms can contain local variable declarations. There are no global variable declarations, local constant or type declarations, or nested subprogram declarations. There are no packages, tasks, exception handling, or generic units.

Statements: Statements can be of the following kinds: assignment, block, if-then-else, for-loop, null, procedure-call, and function-return. The Ada *sequence of statements* is replaced by the *composition of statements*, which has the same syntax but does not restrict execution order. There are no case, while-loop, exit, or goto statements, and no statements that relate to tasks or exception handling.

Data Types: Integer, floating-point, Boolean, and string types are predefined. Array, record, and access types can be defined. There are no character, enumeration, range, fixed-point, unconstrained-array, or variant-record types. There are no subtypes or derived types.

Operators: Logical, relational, and arithmetic operators from Ada can be used. There are strict logical operators, where both operands are always evaluated, and short-circuit forms, where the right operand is only evaluated if necessary. Components can be indexed from arrays and selected from records, and array and record values can be constructed from aggregates of components. Designated objects can be allocated using the **new** operator, with deallocation being the responsibility of the implementation.

2.2 The Single-Assignment Restriction

With Declarative Ada, all variables are *single-assignment variables*, subject to the following restrictions:

- Variables are initialized to a special *undefined* value.
- Evaluation of an undefined variable as an expression suspends until the variable is assigned a value.
- A variable can be assigned a value at most once. It is an execution error if the same variable is assigned a value more than once. As with other execution errors, the result of a program with this error is implementation-dependent.

The single-assignment restriction changes Declarative Ada from a sequential imperative subset of Ada to a parallel declarative programming language.

2.3 Parallel Processes

A program is implicitly executed as a group of *processes*, communicating and synchronizing though shared single-assignment variables. A process is any of the following units of computation:

- The execution of a statement.
- The execution of a composition of statements.
- The evaluation of a variable name.
- The evaluation of an expression.

Processes are *executed in parallel* in the sense that the result of executing a program is equivalent to the result of a fair interleaving of the atomic actions of the individual processes. A program can be considered to be executed on an arbitrarily large number of processors, with each process executed on a separate processor. The result of executing a program is independent of whether processes are actually executed: truly concurrently

on separate processors, interleaved on a single processor, in some acceptable sequential order on a single processor, or a combination of the above. (The only execution requirement is that the fairness rule given in Section 2.6 is satisfied.)

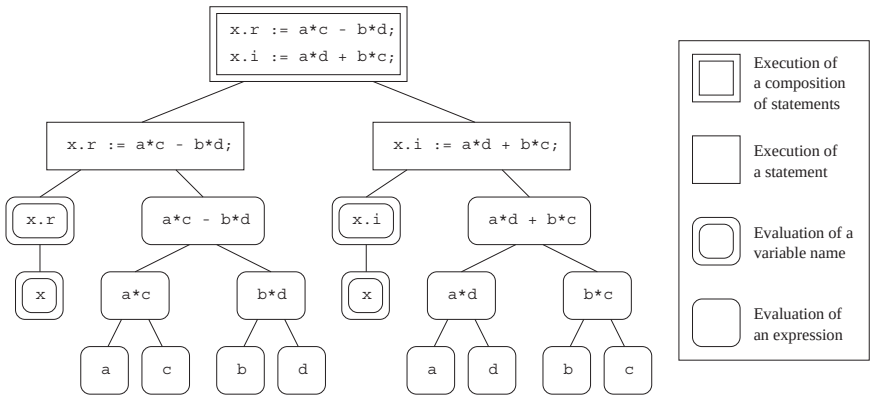

Figure 1: Example of parallel processes in the execution of a composition of statements.

Note: The evaluation of a name on the left-hand side of an assignment statement, or as an **out** or **in out** mode actual parameter is the evaluation of the name as a *variable name*. The evaluation of a name on the right-hand side of an assignment statement, or as an **in** mode actual parameter is the evaluation of the name as an *expression*.

2.4 Process Initiation and Termination

Any non-trivial process implicitly *initiates* other parallel processes, through the execution of enclosed statements and compositions of statements, and the evaluation of enclosed variable names and expressions. A process *terminates* after it has no further actions to perform and all of its subprocesses have terminated. This single mechanism explains parallel execution at all levels of granularity. In this manner, parallelism is implicit in the execution of the following language constructs:

Assignment Statements

variable-name := expression;

The evaluation of the left-hand-side variable name and the evaluation of the right-hand-side expression are parallel processes.

If-Then-Else Statements

if condition **then**
    composition-of-statements
else
    composition-of-statements
end if;

The evaluation of the condition and the execution of the selected composition of statements are parallel processes. The execution of the selected composition of statements cannot be initiated until the evaluation of the condition has yielded a result.

For-Loop Statements

for identifier **in** [**reverse**] lower-bound .. upper-bound **loop**
    composition-of-statements
end loop;

The evaluations of the loop range lower and upper bounds, and the iterations of the execution of the composition of statements are parallel processes. For each iteration, the loop-parameter identifier is the name of a different constant from the loop range. The iterations cannot be initiated until the evaluations of the bounds have yielded results. Whether or not the loop range is specified as **reverse** order has no effect on the meaning of the for-loop statement.

Example: Vector addition:

```
for i in 1 .. N loop
   u(i) := v(i) + w(i);
end loop;
```

Execution of the above program segment computes the `N`-element vector addition and assignment: `u := v + w;`. The `N` executions of the assignment statement are parallel processes.

Procedure-Call Statements

procedure-name(actual-parameter, . . . , actual-parameter);

The evaluations of the actual parameter variable names and expressions, and the execution of the procedure body are parallel processes. The execution of the procedure body can be initiated before the evaluations of the actual parameters have yielded results. The textual order of the parameters in a procedure declaration and its corresponding procedure-call statements has no effect on the meaning of the procedure-call statements.

Example: Matrix multiplication:

```
for i in 1 .. N loop
   for j in 1 .. N loop
      multiply(row(A, i), column(B, j), result(i, j));
   end loop;
end loop;
```

Execution of the above program segment computes the `N`-by-`N` matrix multiplication and assignment: `result := A * B;`. The N^2 executions of the `multiply` procedure-call statement are parallel processes. For each execution of the `multiply` procedure-call statement, the evaluations of the `row(A, i)` and `column(B, j)` function-call expressions, the evaluation of the `result(i, j)` variable name, and the execution of the body of the `multiply` procedure are parallel processes.

Function-Return Statements

return expression;

The evaluation of the returned expression is a parallel process.

Compositions of Statements

statement
...
statement

The executions of the statements are parallel processes. The textual order of the statements has no effect on the meaning of the composition of statements.

Example: Defining an identity matrix:

```
for i in 1 ..  N loop
   for j in 1 ..  i−1 loop A(i, j) := 0; end loop;
   A(i, i) := 1;
   for j in i+1 ..  N loop A(i, j) := 0; end loop;
end loop;
```

Execution of the above program segment assigns the `N`-by-`N` identity matrix to `A`. The `N` executions of the composition of statements enclosed within the outer for-loop are parallel processes. For each execution of the composition of statements, the executions of the assignment statement and the two inner for-loop statements are parallel processes. In total, there are $\texttt{N}^2$ parallel assignment statement processes, one for each `A(i, j)`.

Variable Names

The evaluations of prefix variable names and the evaluations of enclosed expressions are parallel processes.

Example: An indexed component:

```
stack.items(stack.top - 1)
```

The evaluation of the `stack.items` variable name and the evaluation of the enclosed `stack.top - 1` expression are parallel processes.

Example: A selected component:

```
pixel(x, y).color
```

The evaluation of the `pixel(x, y)` variable name is a parallel process.

Expressions

The evaluations of subexpressions are parallel processes.

Example: A strict binary operator:

```
(x + y) * (x - y)
```

The evaluations of the `(x + y)` and `(x - y)` subexpressions are parallel processes. The evaluation of the expression does not yield a result until the evaluations of both subexpressions have yielded results.

Example: A short-circuit binary operator:

```
(x = y) or else (x = z)
```

The evaluation of the `(x = y)` subexpression is a parallel process. If the evaluation of `(x = y)` yields `true`, the evaluation of the expression yields `true`, and the `(x = z)` subexpression is not evaluated. If the evaluation of `(x = y)` yields `false`, the evaluation of the `(x = z)` subexpression is a parallel process.

Example: Functional composition:

```
merge(sort(left), sort(right))
```

The evaluations of the `sort(left)` and `sort(right)` function-call subexpressions, and the execution of the body of the `merge` function are parallel processes.

2.5 Executable and Suspended Processes

Between its initiation and termination, every process is either *executable* or *suspended*. A process becomes suspended when either:

1. The process is the evaluation of a name as an expression, and the value of the name is undefined. In this case, the process becomes executable after the name is assigned a value by another parallel process.
2. To perform any further actions, the process requires values that have not yet been evaluated by its subprocesses. In this case, the process becomes executable after its subprocesses have evaluated the required values.
3. The process has no further actions to perform, but some of its subprocesses have not yet terminated. In this case, the process terminates after all of its subprocesses have terminated.

An executable process remains executable until it is executed. It is impossible for an executing process to change another process from executable to suspended or terminated.

2.6 Fairness

An implementation of the language can schedule processes in any manner such that the following *fairness rule* is satisfied:

From any point in time in a program's execution, every executable process will eventually have some progress made on its execution.

The fairness rule is required to prevent infinite processes from "locking out" other processes. This is a weak form of fairness—there is no requirement to share processing time in any sense "evenly" amongst processes. In a finite program execution with no feedback in data flow, a non-interleaved sequential ordering of processes can always be found that satisfies the fairness rule.

2.7 Communication and Synchronization

Communication and synchronization between parallel processes is implicit: communication is through shared variables, and synchronization is through one process suspending until another process assigns a value to a shared variable. This single mechanism explains communication and synchronization between parallel processes at all levels of granularity. In this manner, the scheduling of processes is automatically constrained by the data flow that occurs during the execution of a program.

For example, in the execution of the following program segment, the assignment statements are independent and can be executed without communication or synchronization:

```
x := 1;
y := 2;
z := 3;
```

In contrast, in the execution of the following program segment, the assignment statements are not independent:

```
x := a; -- #1
y := x; -- #2
z := x; -- #3
```

The executions of the three assignment statements are parallel processes, but statements #2 and #3 suspend until statement #1 assigns a value to `x`.

After x is assigned a value, statements #2 and #3 can be executed without communication or synchronization. The textual order of the statements is not significant. For example, the following program segment has exactly the same meaning:

```
z := x; -- #3
y := x; -- #2
x := a; -- #1
```

As another example, in the execution of the following program segment, the N iterations of the for-loop statement are independent and can be executed without communication or synchronization:

```
for i in 1 .. N loop
   A(i) := i;
end loop;
```

In contrast, in the execution of the following program segment, the N iterations of the for-loop statement are not independent:

```
A(1) := 1;
for i in 2 .. N loop
   A(i) := A(i-1) + 1;
end loop;
```

The execution of the initial assignment statement and the N−1 iterations of the for-loop statement are parallel processes, but assignment to each A(i) suspends until A(i−1) is assigned a value. In this manner, the parallel iterations of the for-loop statement are executed sequentially. The textual order of the statements and the order of the loop range are not significant. For example, the following program segment has exactly the same meaning:

```
for i in reverse 2 .. N loop
   A(i) := A(i-1) + 1;
end loop;
A(1) := 1;
```

More sophisticated communication and synchronization between processes is possible through shared variables of more complex types, e.g., arrays, linked lists, and trees. However, the same simple mechanism of suspension on undefined values explains these interactions. For example, in *producer-consumer* interactions, a *producer* process generates as output a data structure that is used as input by a *consumer* process. The consumer suspends when it attempts to use the value of an undefined component of the data structure, and it resumes execution after the producer assigns a value to that component. It is not required that all the components of the shared data structure be assigned values by the producer before the consumer is executed.

2.8 Determinacy

The results of programs are deterministic in the sense that over all possible executions of the same program with the same input values:

- If execution terminates without errors, it will always terminate without errors, and the output values will always be the same.
- If execution permanently suspends without errors, it will always permanently suspend without errors, and the output values will always be the same.
- If an execution error occurs, an execution error will always occur, though not necessarily always the same execution error.
- If execution is non-terminating and without permanent suspension or errors, it will always be non-terminating and without permanent suspension or errors, and the output values will always be the same.

In this sense, the results of programs are independent of variations in the number of processors, the mapping of processes onto processors, and the scheduling of processes. Such repeatability of results is generally considered to be desirable, particularly for program development, testing, and debugging. For instance, a program can be developed on a single processor, then executed on multiple processors for improved performance with the same results.

Although there can be nondeterminacy in the mapping and scheduling of processes, repeated executions of the same program with the same inputs will always perform the same computations and define the same intermediate and output values. This is a consequence of the single-assignment restriction and the absence of other nondeterministic language constructs. For example, without the single-assignment restriction, consider the parallel execution of the following program segment, beginning in a state where `x` has the value `1`:

```
x := x + 1;
x := x * 3;
```

The possible resulting values of `x` include `2`, `3`, `4`, and `6`.

Another nondeterministic construct that is absent from the language is a *probe* to test whether or not the value of a variable is defined. For example, consider the parallel execution of the following program segment, beginning in a state where the values of `x` and `y` are undefined:

```
x := 1;
if x is defined then y := 1; else y := 2; end if;
```

The resulting value of `y` could be either `1` or `2`.

The addition of nondeterministic constructs from outside the pure language for problems that explicitly require nondeterminacy is discussed in Chapter 5.

3 Example Programs

In this section we present and discuss three different complete Declarative Ada example programs that cover a range of parallel programming techniques and together include most language constructs. A larger collection of example programs is presented and discussed in [10].

3.1 Specifications

An *assertion* is a Boolean expression on the variables of a program. With a sequential language, the specification of a subprogram can be given as pair of assertions: a *precondition* and a *postcondition*. The meaning of a specification of this form is:

> *If the precondition holds in the program state in which the execution of a subprogram-call is initiated, the execution of the subprogram-call will terminate, and the postcondition will hold in the program state in which the execution of the subprogram-call terminates.*

Specifications of this form can be used to: document programs, reason about the correctness of programs, and debug programs (by checking the assertions at run-time). An excellent discussion of the practical use of assertions in software construction is given by Meyer [8, Chapter 7].

With Declarative Ada, a precondition and postcondition pair is not an appropriate specification of a subprogram, because the precondition can be undefined when the execution of a subprogram-call is initiated, and the postcondition can be undefined when the execution of a subprogram-call terminates. Instead, the specification of a subprogram can be given as a different pair of assertions: an *input-condition* and an *output-condition*. The meaning of a specification of this form is:

> *If the input-condition holds in some program state that occurs after the execution of a subprogram-call is initiated, the output-condition will hold in some subsequent program state.*

Input-condition and output-condition assertions are required to be *stable*, in the sense that if an assertion has a defined value in a program state, it will have the same defined value in all subsequent program states. For example, expressions such as "x is defined" are not allowed. All expressions that can be written in Declarative Ada are stable.

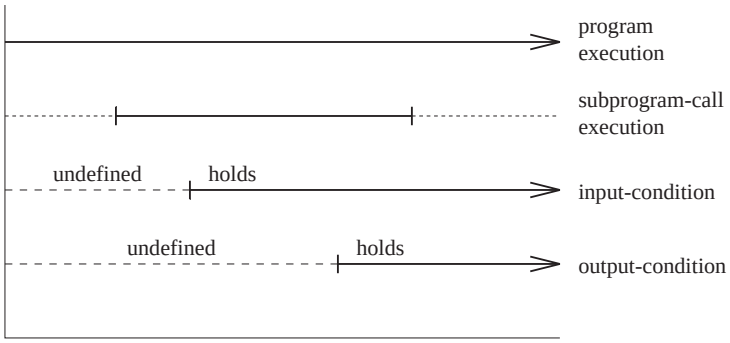

Figure 2: Meaning of the specification of a subprogram given as an input-condition and an output-condition.

Specifications using input-conditions and output-conditions are analogous to specifications using preconditions and postconditions, and have the same uses in documentation, reasoning about correctness, and debugging. In the example programs that follow, input-conditions and output-conditions are given as comments.

3.2 Mergesort

Problem Description

Our first example program is a function, `sort`, that takes as input, `unsorted`, a linked list of integers, and returns a linked list of integers that is a permutation of the elements of `unsorted` in ascending order. The program is implemented using the mergesort algorithm.

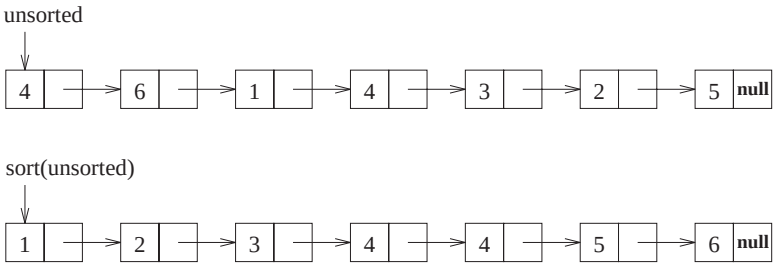

Figure 3: Example of sorting a linked list.

Motivation

The mergesort program demonstrates the following parallel programming techniques:

- Parallelism from composition of statements.
- Parallelism from functional composition.
- Parallelism from recursion.
- Communication and synchronization through linked lists.

It is notable that the mergesort program can be correctly executed as a sequential program without modification.

Program

```
type node;
type list is access node;
type node is record
               head :  integer;
               tail :  list;
             end record;
```

```
procedure split(items :  in list; left, right :  out list) is
-- Input Condition:
--    items is finite and acyclic.
-- Output Condition:
--    left and right combined is a permutation of items, and
--    left and right differ in length by at most one element.
   left_tail, right_tail :  list;
begin
   if items = null or else items.tail = null then
      left  := items;
      right := null;
   else
      split(items.tail.tail, left_tail, right_tail);
      left  := new node'(items.head, left_tail);
      right := new node'(items.tail.head, right_tail);
   end if;
end split;

function merge(left, right :  list) return list is
-- Input Condition:
--    left and right are both finite and acyclic, and
--    left and right are both in ascending order.
-- Output Condition:
--    merge(left, right) is a permutation of left and right combined, and
--    merge(left, right) is in ascending order.
begin
   if left = null then
      return right;
   elsif right = null then
      return left;
   elsif left.head <= right.head then
      return new node'(left.head, merge(left.tail, right));
   else
      return new node'(right.head, merge(left, right.tail));
   end if;
end merge;
```

```
function sort(unsorted :  list) return list is
-- Input Condition:
--    unsorted is finite and acyclic.
-- Output Condition:
--    sort(unsorted) is a permutation of unsorted, and
--    sort(unsorted) is in ascending order.
   left, right :  list;
begin
   if unsorted = null or else unsorted.tail = null then
      return unsorted;
   else
      split(unsorted, left, right);
      return merge(sort(left), sort(right));
   end if;
end sort;
```

Discussion

The `sort` function is implemented using the following recursive algorithm:

- If `unsorted` is the empty list or a list with one element, `sort` returns `unsorted`.
- If `unsorted` has more than one element, `unsorted` is split into `left` and `right` sublists, and `sort` returns the ordered merge of `sort(left)` and `sort(right)`.

By induction on the number of elements of `unsorted`, and from the specifications of `split` and `merge`, this implementation satisfies the specification of `sort`.

In the evaluation of a non-trivial `sort` call, the execution of the `split` call, the evaluation of the `merge` call, and the evaluations of the recursive `sort` calls are parallel processes. The executions of the `split` call and the return statement that evaluates the `merge` call are parallel processes from being in a composition of statements. The recursive `sort` calls are parallel processes from being subexpressions of the `merge` call.

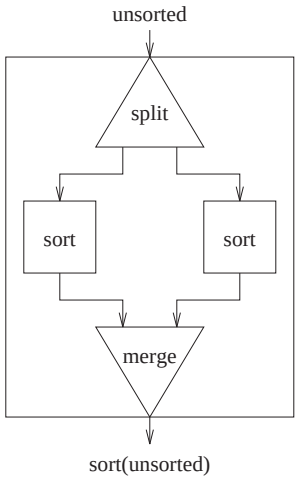

Figure 4: Parallel processes in the evaluation of a non-trivial `sort` call.

Communication and synchronization between the parallel processes in the evaluation of a non-trivial `sort` call is through shared linked lists. Two different kinds of interprocess interaction are present:

1. **Independent processes:** The recursive `sort` processes can be executed independently of each other without communication or synchronization.

2. **Dependent processes:** The recursive `sort` processes are dependent on the output of the `split` process, and the `merge` process is dependent on the output of the recursive `sort` processes. Synchronization is through dependent processes suspending when they attempt to use an undefined input element, and resuming execution after the element is defined. Parallelism of this kind—where the producer and consumer of a stream of values are executed in parallel—is often called *pipeline parallelism.*

Because the `sort` function is recursive, the complete evaluation of a `sort` call consists of a network of parallel `split`, recursive `sort`, and `merge` processes. Many different schedulings of processes are allowed. At one extreme,

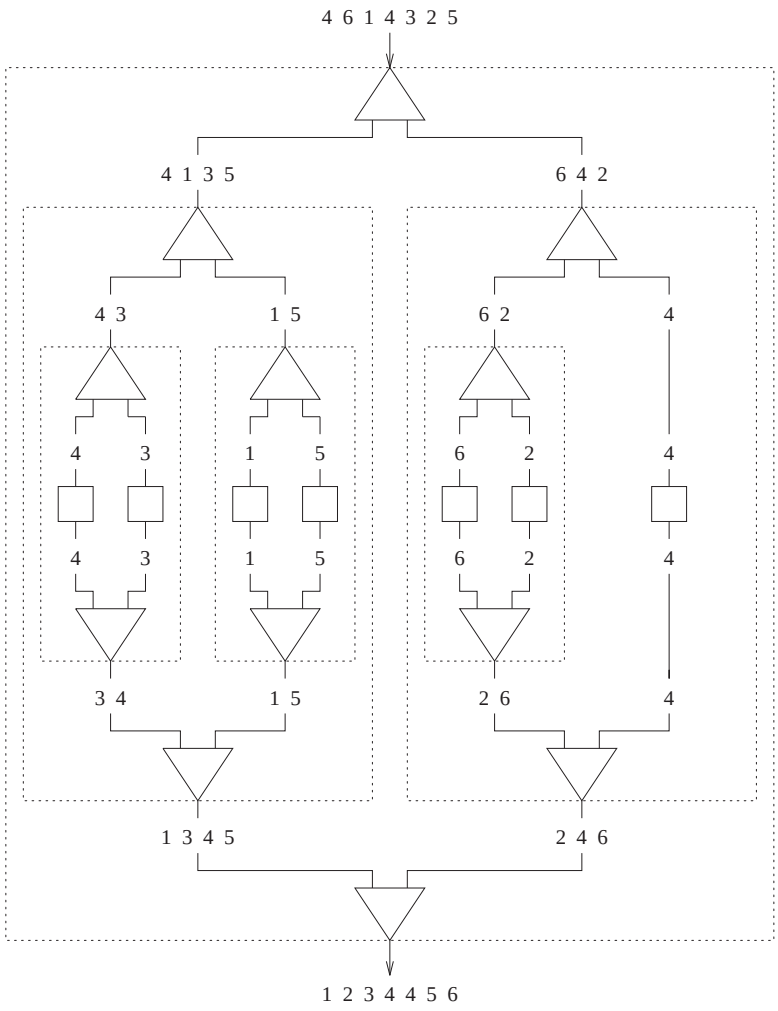

Figure 5: Example of the network of parallel processes for a complete `sort` call.

the program could be executed as a conventional sequential program: if processes are executed in textual sequence without interleaving, variables will be assigned values before they are used in expressions, and processes will execute to termination without suspension. At the other extreme, each process could be executed on a separate processor, with concurrent execution constrained only by data flow.

Although this example program sorts linked lists, exactly the same algorithm could be used to sort arrays, with the `sort` function taking as input an unsorted array value, and returning a sorted array value. Sorting an array "in place" is inconsistent with the single-assignment restriction.

3.3 The All-Points Shortest-Path Problem

Problem Description

Our second example program is a function, `shortest_paths`, that takes as input, `edges`, the edge-weight matrix of a weighted directed graph with vertices V_1 .. V_N, and returns a matrix of the shortest-length paths between all pairs of vertices in the graph. The graph is required to have no cycles of negative length, and the weight of the edge from a vertex to itself is required to be zero, for all vertices. The program is implemented using the Floyd-Warshall algorithm.

Motivation

The all-points shortest-path program demonstrates the following parallel programming techniques:

- Parallelism from for-loop statements.
- Communication and synchronization through arrays.

As with the mergesort program, the all-points shortest-path program can be correctly executed as a sequential program without modification.

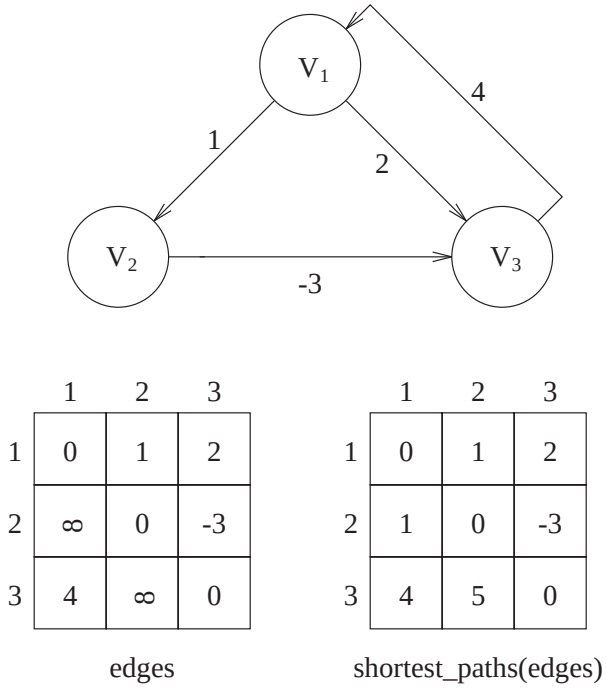

Figure 6: Example of computing the shortest paths of a weighted directed graph.

Program

```
N : constant integer := 10;

type path_lengths     is array(1 ..  N, 1 ..  N) of integer;
type all_path_lengths is array(0 ..  N) of path_lengths;

function min(x, y :  integer) return integer is
begin
   if x < y then return x; else return y; end if;
end min;
```

```
function shortest_paths(edges :  path_lengths) return path_lengths is
-- Input Condition:
--    N ≥ 1, and
--    for some weighted directed graph G, with vertices V1 .. VN:
--       ∀ i ∈ 1..N, j ∈ 1..N : edges(i, j) = weight of edge from Vi to Vj, and
--    G has no cycles of negative length, and
--    ∀ i ∈ 1..N : edges(i, i) = 0.
-- Output Condition:
--    ∀ i ∈ 1..N, j ∈ 1..N :
--       shortest_paths(edges)(i, j) = length of shortest path from Vi to Vj.
   paths :  all_path_lengths;
begin
   paths(0) := edges;
   for k in 1 ..  N loop
      for i in 1 ..  N loop
         for j in 1 ..  N loop
            paths(k)(i, j) := min(paths(k−1)(i, j),
                                  paths(k−1)(i, k) + paths(k−1)(k, j));
         end loop;
      end loop;
   end loop;
   return paths(N);
end shortest_paths;
```

Discussion

The implementation of the `shortest_paths` function computes an array of intermediate results, `paths`, satisfying the following specification:

$\forall$ `k` $\in$ `0..N`, `i` $\in$ `1..N`, `j` $\in$ `1..N` :
`paths(k)(i, j)` = length of shortest path from V_i to V_j with intermediate vertices only in $V_1..V_k$.

The algorithm for computing the `paths` array and the result of the `shortest_paths` function is as follows:

- `paths(0)` = `edges`.

  `edges(i, j)` is the shortest path from V_i to V_j with no intermediate vertices.

- $\forall$ k $\in$ 1..N, i $\in$ 1..N, j $\in$ 1..N :
  paths(k)(i, j) = min(paths(k−1)(i, j),
  paths(k−1)(i, k) + paths(k−1)(k, j)).

  By induction on the number of intermediate vertices, k, this relationship between components satisfies the specification of paths.

- shortest_paths returns paths(N).

  paths(N)(i, j) is the shortest path from V_i to V_j with no restriction on intermediate vertices.

In the evaluation of a shortest_paths call, the executions of the initial array assignment statement, the nested for-loop statement, and the return statement are parallel processes. From the for-loop statement, the $O(N^3)$ executions of the assignment statement are parallel processes, each assigning a value to a different paths(k)(i, j). However, concurrent execution is constrained by the dependence of the assignment to paths(k)(i, j) on paths(k−1)(i, j), paths(k−1)(i, k), and paths(k−1)(k, j). At most $O(N^2)$ assignment statement processes are executable at any point in the execution of the program. Synchronization is through dependent processes suspending when they attempt to use an undefined array component, and resuming execution after the component is defined.

As with the mergesort program, many different schedulings of processes are allowed, including conventional sequential execution.

3.4 Hamming's Problem

Problem Description

Our third example program is a function, hamming, that takes as input, n, a non-negative integer, and returns a linked list, in ascending order, of the first n integers of the form $2^i \times 3^j \times 5^k$, for all integers $i \geq 0, j \geq 0, k \geq 0$. The program is implemented as a network of processes with feedback in the communication between processes.

Motivation

The Hamming's problem program demonstrates that Declarative Ada is more than a restriction on a sequential subset of Ada. Unlike the preceding mergesort and all-points shortest-path programs, the following Hamming's problem program cannot be correctly executed as conventional sequential program, because of feedback in data flow.

Terminating programs without feedback in data flow can always be textually transformed—by reordering statements and expressions—so that conventional sequential execution is an allowed scheduling of processes. For these programs, parallel execution is possible but not necessary. Some nonterminating programs and programs with feedback in data flow cannot be textually transformed into correct sequential programs. For these programs, parallel execution is necessary.

Program

```
type node;
type list is access node;
type node is record
                head :  integer;
                tail :  list;
             end record;

function multiply(m :  integer; numbers :  list) return list is
-- Input Condition:
--    numbers is finite and acyclic.
-- Output Condition:
--    multiply(m, numbers) = a list of the elements of numbers, each multiplied by m.
begin
   if numbers = null then
      return null;
   else
      return new node'(m * numbers.head, multiply(m, numbers.tail));
   end if;
end multiply;
```

```
function merge(input1, input2 :  list) return list is
-- Input Condition:
--    input1 and input2 are both finite and acyclic, and
--    input1 and input2 are both in ascending order, and
--    input1 and input2 both have no duplicate elements.
-- Output Condition:
--    merge(input1, input2) = a list, in ascending order and without duplicates,
--                            of the combined elements of input1 and input2.
begin
   if input1 = null then
      return input2;
   elsif input2 = null then
      return input1;
   elsif input1.head < input2.head then
      return new node'(input1.head, merge(input1.tail, input2));
   elsif input2.head < input1.head then
      return new node'(input2.head, merge(input1, input2.tail));
   elsif input2.head = input1.head then
      return new node'(input1.head, merge(input1.tail, input2.tail));
   end if;
end merge;

function truncate(n :  integer; numbers :  list) return list is
-- Input Condition:
--    n ≥ 0 and numbers is finite and acyclic.
-- Output Condition:
--    truncate(n, numbers) = a list of the first k elements of numbers,
--                           where k = min(n, length of numbers)).
begin
   if n = 0 or  numbers = null then
      return null;
   else
      return new node'(numbers.head, truncate(n - 1, numbers.tail));
   end if;
end truncate;
```

```
function hamming(n :  integer) return list is
-- Input Condition:
--    n ≥ 0.
-- Output Condition:
--    hamming(n) = a list, in ascending order, of the first n integers of the
--                 form 2^i × 3^j × 5^k, for all integers i ≥ 0, j ≥ 0, k ≥ 0.
   times_2, times_3, times_5, merged, result :  list;
begin
   times_2 := multiply(2, result);
   times_3 := multiply(3, result);
   times_5 := multiply(5, result);
   merged := merge(times_2, merge(times_3, times_5));
   result := truncate(n, new node'(1, merged));
   return result;
end hamming;
```

Discussion

The implementation of the `hamming` function is derived from the following recursive definition of the infinite sequence of integers of the form $2^i \times 3^j \times 5^k$, for all integers $i \geq 0, j \geq 0, k \geq 0$:

- 1 is in the sequence.
- If `h` is in the sequence, $2\times$`h`, $3\times$`h`, and $5\times$`h` are also in the sequence.

The `hamming` function returns the `result` list from the network of parallel processes in Figure 7. The first element of `result` is 1, and the subsequent elements are computed through feedback of the ordered merge of `multiply(2, result)`, `multiply(3, result)`, and `multiply(5, result)`. Duplicate elements are removed as part of `merge`. The `truncate` process limits the computation to `n` elements.

Communication between the processes is through shared linked lists. The method of synchronization is the same as for the mergesort program: processes suspend when they attempt to use an undefined input element, and resume execution after the element is defined.

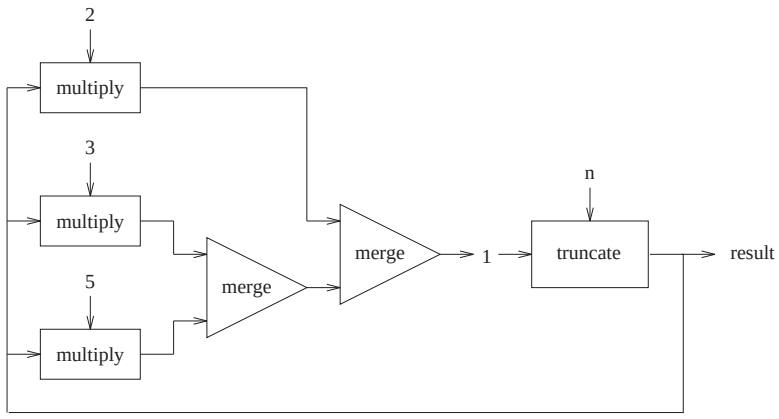

Figure 7: Process network for Hamming's problem.

4 Sequential Input and Output

In this section we describe how deterministic sequential input and output is performed with Declarative Ada using *sequential composition*. We explain that sequential input and output files are a specific instance of multiple-assignment variables, and that sequential composition is required for operations on multiple-assignment variables. This leads us to describing how an extended version of Declarative Ada could integrate single-assignment variables and general multiple-assignment variables.

4.1 Input and Output

Input and output operations are performed using calls to predefined procedures: `get`, `put`, and `new_line`. Input is read from the program's standard input file and output is written to the program's standard output file. For example, the following program segment reads an integer and writes whether or not it is even:

```
get(number);
if number mod 2 = 0 then
   put("yes");
else
   put("no");
end if;
```

The result of parallel execution of input and output statements can be nondeterministic. For example, consider parallel execution of the following program segment:

```
put("one");
put("two");
```

The output written is unpredictable: it could be any of `onetwo`, `twoone`, `otnweo`, `one`, `two`, etc.

For this reason, parallel execution of input and output statements is restricted in the following manner:

- It is an execution error if more than one parallel subprocess of a process executes an output statement.
- It is an execution error if more than one parallel subprocess of a process executes an input statement.

As with other execution errors, the result of a program with this error is implementation-dependent.

4.2 Sequential Composition

Deterministic sequential input and output can be performed using *sequential composition*. Sequential composition occurs in two language constructs: sequential compositions of statements and sequential for-loop statements.

Sequential Compositions of Statements

sequential
statement
...
statement

The executions of the statements occur sequentially in their textual order. Execution of each statement terminates before execution of the next statement is initiated.

Example: Sequential input and output:

```
sequential
get(n);
if n < 0 then
   put("Illegal input.");
else
   sequential
   put(n); put("!  = "); put(factorial(n));
end if;
new_line;
```

Sequential compositions of statements can be used wherever parallel compositions of statements can be used: within blocks, if-then-else statements, and for-loop statements. Parallel and sequential compositions of statements can be nested arbitrarily.

Sequential For-Loop Statements

for identifier **in** [**reverse**] range **sequential loop**
    composition-of-statements
end loop;

The iterations of the execution of the composition of statements occur sequentially in the order of the loop range. Each iteration terminates before the next iteration is initiated.

Example: Output of a two-dimensional array:

```
for i in 1 .. N sequential loop
   sequential
   for j in 1 .. N sequential loop
      sequential
      put(A(i, j)); put(" ");
   end loop;
   new_line;
end loop;
```

Whether the for-loop statement is parallel or sequential does not affect whether the enclosed composition of statements is parallel or sequential. Both parallel and sequential compositions of statements can be enclosed by both parallel and sequential for-loop statements.

4.3 Multiple-Assignment Variables

The reason that the result of parallel execution of input and output statements can be nondeterministic is that the program's input file and output file are implicitly-declared multiple-assignment variables. Every read operation *updates* the same input file, and every write operation *updates* the same output file. (Although a read operation does not update the components of the input file, it updates the position from which the next component will be read.) The result of multiple updates to the same file is dependent on the order in which they are performed. Since the execution order of parallel processes is nondeterministic, the result of parallel input and output operations is also nondeterministic.

Currently the implicitly-declared input file and output file are the only multiple-assignment variables allowed in a program. However, an extended version of Declarative Ada could integrate single-assignment variables and general multiple-assignment variables. Parallel execution of operations on multiple-assignment variables would be restricted: it would be an execution error if an assignment to a multiple-assignment variable was executed in parallel with any other operation on the same variable. Sequential composition could be used to update multiple-assignment variables sequentially, in the same manner as it is currently used to perform input and output operations

sequentially.

For example, the following function, `multiply`, in an extension of Declarative Ada, returns the single-assignment product of two single-assignment matrices. All variables are single-assignment unless specified as **multiple**, and all compositions of statements and for-loop statements are parallel unless specified as **sequential**:

```
type matrix is array(1 .. N, 1 .. N) of float;

function multiply(A, B : matrix) return matrix is
   result : matrix;
begin
   for i in 1 .. N loop
      for j in 1 .. N loop
         declare
            sum : multiple float;
         begin
            sequential
            sum := 0.0;
            for k in 1 .. N sequential loop
               sum := sum + A(i, k) * B(k, j);
            end loop;
            result(i, j) := sum;
         end;
      end loop;
   end loop;
   return result;
end multiply;
```

In the evaluation of a `multiply` call, the $\mathtt{N}^2$ executions of the inner block are parallel processes, each assigning a value to a different `result(i, j)`. For each execution of the inner block, `sum` is a multiple-assignment variable that sequentially accumulates the inner-product of the `i`th row of `A` and the `j`th column of `B`. With only single-assignment variables and parallel composition, the inner-product computation would require a separate recursive function.

PCN and CC++ [3] are languages that integrate parallel and sequential composition and single-assignment and multiple-assignment variables.

5 Reactive Programs

In this section we discuss *reactive* programs: programs that interact with their environment on an ongoing basis. For reactive programming with Declarative Ada, we propose the use of a *fair merge* procedure from outside the pure language, to provide controlled nondeterminacy.

5.1 Reactive Programs and Nondeterminacy

Reactive programs are programs that interact with their environment on an ongoing basis. Examples of reactive programs include: computer operating systems, programs that monitor and control physical systems, and programs that interact directly with input and output devices.

Reactive programs are typically required to wait for any one of many different conditions to hold, then respond appropriately within finite time. Often there is no guarantee that any particular condition will eventually hold. Implementation of a wait on many different conditions requires some kind of nondeterministic language construct. A deterministic wait can suspend forever, waiting for one particular condition to hold. Language constructs that provide nondeterminacy include: probes, nondeterministic choice, and fair merge.

5.2 Fair Merge

For reactive programming with Declarative Ada, we propose the use of a *fair merge* procedure from outside the pure language. Fair merge as the only nondeterministic language construct isolates the nondeterminacy in a program in a few obvious and well-defined `merger` procedure call statements. All components of the program apart from the `merger` calls are deterministic. We believe that "controlling" nondeterminacy in this manner is helpful in reasoning, testing, and debugging.

The fair merge procedure has the following interface:

```
type node;
type list is access node;
type node is record
               head :  integer;
               tail :  list;
             end record;

procedure merger(left, right :  in list; merged :  out list);
```

Informally, the specification of `merger` is as follows:

- `merged` is an interleaving of the elements of `left` and `right`.
- The order of the elements of `left` and `right` is preserved in `merged`.
- All elements of `left` and `right` eventually appear in `merged`, even if `left` or `right` are infinite lists.

A more formal specification of fair merge—in the context of PCN—is given by Chandy and Taylor [4, Chapter 14].

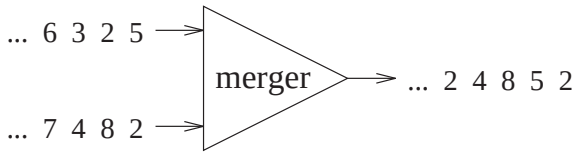

Figure 8: Example of fair merge.

Any implementation that satisfies the specification of `merger` is nondeterministic. The decision to copy elements from one input list when the other input list is undefined requires a nondeterministic probe of the state of the undefined input list.

Although we specify the `merger` procedure as merging two lists of integers, variants of the `merger` procedure could merge lists of other types or more than two lists. The method by which a `merger` procedure is imported into a program is implementation-dependent.

5.3 Example: Raising an Alarm

Our first reactive example is a process that monitors two input streams and "raises an alarm" if any element of either stream exceeds a threshold value. The alarm must be raised within finite time, regardless of the state of the other input stream.

The procedure, `raise_alarm`, takes as input an integer, `threshold`, and two infinite lists of integers, `left` and `right`, and returns as output a Boolean variable, `alarm`. If any element of either input list has a value greater than `threshold`, `alarm` is assigned `true` within finite time. The program uses the `list` type and `merger` procedure that are defined in Section 5.2.

```
procedure check(threshold :  in  integer;
                input     :  in  list;
                alarm     :  out Boolean ) is
begin
   if input.head > threshold then
      alarm := true;
   else
      check(threshold, input.tail, alarm);
   end if;
end check;

procedure raise_alarm(threshold   :  in  integer;
                      left, right :  in  list;
                      alarm       :  out Boolean ) is
   merged :  list;
begin
   merger(left, right, merged);
   check(threshold, merged, alarm);
end raise_alarm;
```

The fair merge procedure, `merger`, is used to merge `left` and `right` into a single list, `merged`. The deterministic `check` procedure assigns `true` to `alarm` if any element of `merged` has a value greater than `threshold`. By the specification of `merger`, all elements of `left` and `right` will appear in `merged` in finite time. Therefore, if any element of either `left` or `right` has a value greater than `threshold`, `alarm` is assigned `true` within finite time.

An implementation of `raise_alarm` with fair merge replaced by a deterministic merge does not satisfy the specification. A deterministic merge will not

continue to copy the elements from one input list when the other input list is undefined.

5.4 Example: Distributed Dining Philosophers

Our second reactive example is the distributed dining philosophers problem. We describe the problem and give an outline of a solution that uses fair merge, but do not present any program text here. The complete Declarative Ada program is presented and discussed in [10, Section 5].

Problem Description

The problem is to implement a distributed operating system that controls access to a shared resource amongst a group of independent user processes called *clients*. Each of the clients repeatedly requests access to the resource, waits to be granted access to the resource, uses the resource, then relinquishes access to the resource. Underlying the system is an undirected graph with vertices representing clients.

The behavior of the clients must satisfy the following requirement:

- A client that has been granted access to the resource must eventually relinquish access to the resource.

Subject to the clients satisfying the preceding requirement, the implementation of the distributed operating system must satisfy the following requirements:

- **Safety:** Clients that are neighbors in the underlying graph cannot be granted simultaneous access to the resource.
- **Progress:** A client that requests access to the resource must eventually be granted access to the resource.

Implementation Outline

Our implementation of the distributed operating system consists of a network of identical operating system processes called *servers*. There is one server for every client. Servers communicate with their own client and with the servers of neighboring clients. With the system in Figure 9, clients 2 and 3 can be granted simultaneous access to the resource, but no other client can be granted simultaneous access to the resource with either of clients 1 or 4.

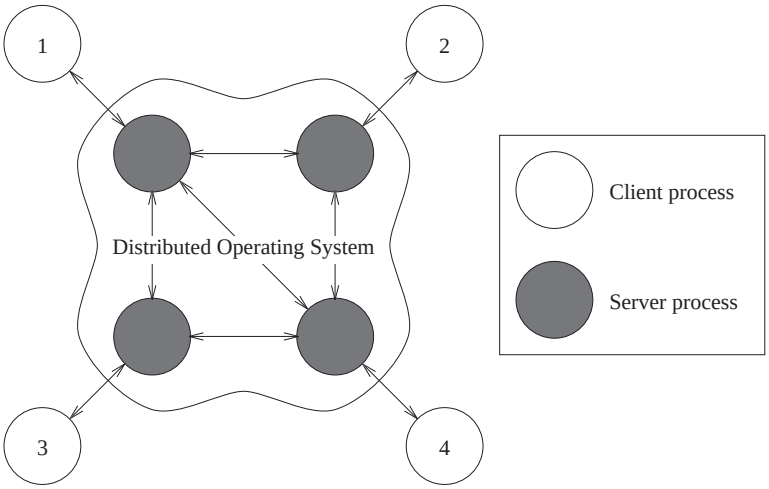

Figure 9: Example of a distributed dining philosophers system.

Each server takes as input the fair merge of message streams from its client and its neighboring servers, and returns as output message streams to its client and its neighboring servers. The only nondeterministic processes in the entire distributed operating system are the fair merge processes.

An implementation with fair merge replaced by a deterministic merge does not satisfy the progress requirement. A deterministic merge will not continue to copy the messages from the other input streams when one input stream is undefined. For example, if one client does not ever request access to the resource, all connected servers will eventually suspend.

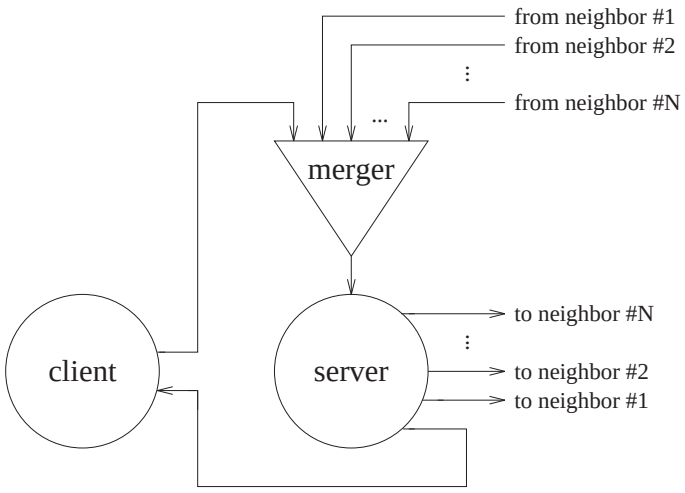

Figure 10: Communication between servers and clients using fair merge.

6 Implementation Issues

In this section we identify some of the issues involved in translating a Declarative Ada source program into an efficient object program. At the time of writing, an efficient implementation of Declarative Ada has not been developed.

6.1 Target Architectures

Declarative Ada is suited to implementation on the following classes of target architecture:

- Single processor.
- Multiple processor, single address space.
- Multiple processor, multiple address space.

Declarative Ada is also suited to implementation on some experimental target architectures such as dataflow machines.

6.2 Single-Assignment Variables

Implementation of single-assignment variables is straightforward:

- Each variable has a tag or special value that indicates it is undefined.
- Each undefined variable has a list of suspended processes.
- When a process suspends on an undefined variable, it is added to that variable's list of suspended processes.
- When a variable is assigned a value, the processes in that variable's list of suspended processes are made executable.

Synchronization of concurrent operations on the same single-assignment variable is a variation of the readers and writers problem [5].

For architectures with multiple address spaces, different copies of the same variable can be stored in arbitrarily many different address spaces. After a variable is assigned a value in one address space, the variable must eventually be assigned the same value in the other address spaces. Since a variable can be assigned a value at most once, the only inconsistency that can occur is that a variable has a value in one address space and is still undefined in another address space. Processes suspend when they attempt to use the value of an undefined variable. Therefore, no problems are caused by this temporary inconsistency.

The overhead of checking whether single-assignment variables are defined can be reduced using compile-time data flow analysis to remove redundant checks.

6.3 Controlling Parallelism

Most programs implicitly express more fine grain parallelism than can be efficiently exploited on any practical architecture. Straightforward implementation of these programs, with a separate thread of control for every

parallel process, will devote considerably more memory and processing time to process management than to useful computation. For efficiency, a source program that contains excessive fine grain parallel composition must be transformed into an equivalent object program containing a judicious combination of parallel and sequential composition.

There are two separate issues involved in this transformation:

1. Recognizing which parallel compositions can be correctly transformed into sequential compositions.
2. Of those parallel compositions that can be correctly transformed into sequential compositions, deciding which parallel compositions are more efficiently executed as sequential compositions.

Recognizing which parallel compositions can be correctly transformed into sequential compositions is a difficult problem. Compile-time data flow analysis is a useful tool, but cannot completely solve the problem.

Deciding which parallel compositions are more efficiently executed as sequential compositions is another difficult problem. There are two aspects to the problem:

1. For a particular architecture, it is inefficient to support parallel processes that are too fine grained.
2. In some cases, even for coarse grain processes, data dependencies between processes make sequential composition more efficient than parallel composition.

The difficulty is determining the granularity of processes and the data dependencies between processes. Again, compile-time data flow analysis is a useful tool, but cannot completely solve the problem. Feedback of profiling information from previous executions into the compiler is another useful tool.

6.4 Controlling Copying

Many programs repeatedly copy large data structures with only a few components changed. Straightforward implementation of these programs will require considerably more memory and processing time than equivalent imperative programs in which components of data structures are updated in place. For efficiency, a source program that contains excessive copying of large single-assignment data structures must be transformed into an equivalent object program containing multiple-assignment data structures and components updated in place.

Sophisticated compile-time data flow analysis is necessary to perform these transformations. However, we take heart from the results reported by Skedzielewski [9, pages 151–152] for an implementation of the parallel functional language Sisal:

> *The latest Sisal compiler developed at Colorado State University analyzes programs ... to minimize storage usage. These results show that array copying can be eliminated in nearly all of the cases in the benchmarks studied ...*

The transformations of parallel composition into sequential composition and of single-assignment variables into multiple-assignment variables are obviously interrelated.

6.5 Process Mapping

Processes must be mapped onto processors for execution. Different process mapping strategies are effective for different target architectures. Process mapping can be decided at compile-time or at run-time. Processes can be mapped permanently onto one processor or move between processors dynamically during execution.

All process mapping strategies have the following two concerns:

1. **Load balancing:** To minimize processor idle-time, workload must be shared evenly amongst the processors.

2. **Locality:** To minimize the cost of interprocessor communication, processes with heavy interactions must be mapped onto the same processor or onto processors in the same locality.

A discussion of process mapping strategies is given by Chandy and Taylor [4, Chapter 7].

For some architectures, efficient process mapping decisions can be made automatically. For example, for a multiprocessor with all processors having fast uniform access to a single address space, an efficient mapping strategy may be to map and schedule dynamically from a single pool of executable processes. However, for some architectures, it is difficult to make efficient process mapping decisions automatically. For example, for a network of workstations, with slow non-uniform communication costs between processors, locality may be so important that efficient automatic mapping is impossible.

For architectures that are not suited to automatic process mapping, the programmer can be given explicit control of process mapping by allowing process mapping *annotations* to be embedded in programs. There are two kinds of process mapping annotations:

1. **Process annotations:** Process initiation constructs are annotated, directly or indirectly specifying the mapping of processes onto processors.

2. **Data annotations:** Data declarations are annotated, specifying the distribution of the data across a distributed address space. Processes are mapped according to the data that they access.

Annotations are not part of the syntax of the programming language. They affect only the performance, not the results or correctness, of programs. A description of process mapping annotations for PCN is given by Foster and Tuecke [6, Section 10]. A description of process mapping and scheduling annotations for the functional language Haskell is given by Hudak [7].

6.6 Manual Optimization

The optimizations that we discussed in the preceding sections—introduction of sequential composition to control parallelism, introduction of multiple-assignment variables to control copying, and automatic mapping of processes to processors—require a very sophisticated optimizing compiler. An interim approach is to allow the programmer to use explicit sequential composition, multiple-assignment variables, and process mapping annotations for efficiency. This is the approach taken by PCN and CC++.

A key question in the compilation of declarative languages is how much optimization can be performed automatically, and how much optimization must always rely on the programmer. We do not know what the answer to this question will be in the long-term.

7 Conclusion

Declarative Ada is a programming language that integrates:

- Subprograms, statements, data types, and operators from Ada.
- Deterministic parallel declarative programming, using single-assignment variables.
- Sequential input and output from within parallel programs, using sequential composition.
- Nondeterministic reactive programming, using a fair merge procedure from outside the pure language.

The success of this integration demonstrates that, although there are fundamental differences between declarative programming and imperative programming, most of the issues of programming language design are independent of these differences.

Our programming experience with Declarative Ada convinces us that, for many problems, declarative programming is a practical and useful approach to parallel programming. Because of the high level of abstraction from

the details of parallel execution, many parallel declarative programs are less complicated and more portable than equivalent parallel imperative programs. Declarative Ada presents the important ideas of parallel declarative programming in a form that is accessible to the large body of programmers who are unfamiliar and uncomfortable with the esoteric mathematically-inspired notations of most other parallel declarative languages.

Our programming experience with Declarative Ada has also shown us that declarative programming does not provide the best solution to all problems. For some problems, multiple-assignment variables, explicit parallel processes, and explicit communication and synchronization between processes allow for solutions that are more elegant or more efficient. For this reason, we are enthusiastic about the prospect of expanding the integration of declarative programming and imperative programming in Declarative Ada.

We propose the development of a language that integrates:

- The full Ada language, including parallelism from the Ada tasking model.
- Single-assignment variables and multiple-assignment variables, with the default being multiple-assignment for compatibility with standard Ada.
- Parallel composition and sequential composition, with the default being sequential composition for compatibility with standard Ada.

This language would be a minimal extension of Ada. Yet it would elegantly and fully integrate declarative programming and imperative programming within a practical language for large-scale software engineering.

8 Acknowledgments

Thank you to my advisor, Mani Chandy, for his guidance and support throughout all phases of this research. Also thank you to the other members of the compositional systems research group at Caltech, Adam Rifkin, Berna Massingill, Carl Kesselman, Paul Sivilotti, and Ulla Binau, for their suggestions and proofreading.

Many of the ideas in Declarative Ada are derived from ideas in PCN.

This research was supported in part by Air Force Office of Scientific Research grant AFSOR-91-0070.

References

[1] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual.* ANSI/MIL-STD-1815A-1983. Springer Verlag, Berlin, 1983.

[2] Grady Booch. *Software Engineering with Ada.* Benjamin/Cummings, Menlo Park, California, 1983.

[3] K. Mani Chandy and Carl Kesselman. *CC++: A Declarative Concurrent Object Oriented Programming Language.* CS-TR-92-01. Computer Science Department, California Institute of Technology, 1992.

[4] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett, Boston, 1992.

[5] P. J. Courtois, F. Heymans, and D. L. Parnas. *Concurrent Control with "Readers" and "Writers".* Communications of the ACM, Volume 14, Number 10, October 1971. Pages 667–668.

[6] Ian Foster and Steven Tuecke. *Parallel Programming with PCN.* ANL-91/32 Rev.2. Argonne National Laboratory, Argonne, Illinois, 1993.

[7] Paul Hudak. *Para-Functional Programming in Haskell.* In *Parallel Functional Languages and Compilers.* Edited by Boleslaw K. Szymanski. Addison-Wesley, Reading, Massachusetts, 1991. Pages 159–196.

[8] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, Hemel Hempstead, 1988.

[9] Stephen K. Skedzielewski. *Sisal.* In *Parallel Functional Languages and Compilers.* Edited by Boleslaw K. Szymanski. Addison-Wesley, Reading, Massachusetts, 1991. Pages 105–157.

[10] John Thornley. *A Collection of Declarative Ada Example Programs.* CS-TR-93-05. Computer Science Department, California Institute of Technology, 1993.

[11] John Thornley. *An Implementation of the Programming Language Declarative Ada.* CS-TR-93-06. Computer Science Department, California Institute of Technology, 1993.

[12] John Thornley. *The Programming Language Declarative Ada Reference Manual.* CS-TR-93-04. Computer Science Department, California Institute of Technology, 1993.